ALGONQUIN COLLEGE

ASSIGNMENT 1 SPECIFICATIONS[1] - BUFFER

General View

Due Date: Prior or on **February 6th 2021 (midnight)**

- **1st week late submission** (until February 13th midnight): **30%** off.
- **2nd week late submission** (until February 20th midnight): **50%** off.
- **Later: 100%** off.

Earnings: **5%** of your course grade

Purpose: **Programming and Using Dynamic Structures (buffers) with C**

❖ This is a review of and an exercise using C coding style, programming techniques, data types and structures, memory management, and simple file input/output. It will give you a better understanding of the type of internal data structures used by a simple compiler you will be building this semester. This assignment will be also an exercise on "*excessively defensive programming*".

❖ You are required to write functions that should be "overly" protected and should not abruptly terminate or "**crash**" at run-time due to invalid function parameters, erroneous internal calculations, or memory violations. To complete the assignment, you should fulfill two tasks: ***Task 1: Buffer Implementation*** and ***Task 2: Testing the Buffer***.

❖ The current version of code requires *Camel Code style*. Use it appropriately.

Buffer (before start coding)

Here are some tips (not exactly in a "logical" sequence of steps), with some ideas to help you during the development of A1 (Buffer):

❖ Please read the **Assignment Submission Standard and Assignment Marking Guide** (under "Assignments > Standards" section).

[1] Adapted from resources developed by Prof. Svillen Ranev (Algonquin College, 2019)

- ❖ Be sure that you have configured appropriately the environment on Visual Studio 2019 Community Edition (see **ProjectVisualStudio_v2019** under "Lecture Materials > Development Notes")
- ❖ To do this activity, you need to do the following steps:
  - Download the files (see "**A1W21.zip**" under Assignments > Assignment 1).
  - In your Project, import the files in "a1_code" folder (including the main Program – `evalBuffer.c`.
  - Remember to adjust the files (including names) in your project.

Task 1: Buffer implementation

- ❖ In the incomplete code ("`buffer_incompleter.h`" and "`buffer_incomplete.c`"), check all **TODO** comments.

1.1. BUFFER STRUCTURE – `buffer.h`

Note 1: Remembering Buffers

***Buffers** are often used when developing compilers because of their efficiency (see page 111 of your textbook).*

*The buffer implementation is based on two associated data structures: 1) **Buffer Entity** (or **Buffer Handle**) and, 2) **Array of characters** (the actual character buffer).*

*Both structures are to be created "**on demand**" at run time, that is, they are to be allocated dynamically.*

The Buffer Descriptor or Buffer Handle - the names suggest the purpose of this buffer control data structure. It contains all the necessary information about the array of characters, including a pointer to the beginning of the character array location in memory, the current size, the next character entry position, the increment factor, the operational mode and some additional parameters.

- ❖ You are to implement a buffer that can operate in three different modes: a "**fixed-size**" buffer, an "**additive self-incrementing**" buffer, and a "**multiplicative self-incrementing**" buffer.

The following structure declaration must be used to implement the Buffer:

```
typedef struct Buffer {
    char* content;      /* pointer to the beginning of character array (character buffer) */
    short size;         /* current dynamic memory size (in bytes) allocated to buffer */
    char increment;     /* character array increment factor */
    char mode;          /* operational mode indicator*/
    short addCOffset;       /* the offset (in chars) to the add-character location */
    short getCOffset;       /* the offset (in chars) to the get-character location */
    short markOffset;       /* the offset (in chars) to the mark location */
    unsigned short flags;  /* contains character array reallocation and end-of-buffer flag */
} bStructure, *bPointer;
```

Where:

- ***content*** is the pointer that indicates the beginning of useful information (loaded from a source file).
- ***size*** is the current total size (measured in bytes) of the memory allocated for the character array by ***malloc()/realloc()*** functions. In the text below it is referred also as *current size*. It is whatever value you have used in the call to ***malloc()/realloc()*** that allocates the storage pointed to by ***content***.
- ***increment*** is a buffer increment factor. It is used in the calculations of a new buffer **size** when the buffer needs to grow. The buffer needs to grow when it is full but still another character needs to be added to the buffer. The buffer is full when ***addCOffset*** measured in bytes is equal to ***size*** and thus all the allocated memory has been used. The ***increment*** is only used when the buffer operates in one of the "*self-incrementing*" modes.
    - In "***a**dditive self-incrementing*", mode it is a positive integer number in the **range from 1 to 255** and represents **directly the increment** (measured in characters) that must be added to the current size every time the buffer needs to grow.
    - In "***m**ultiplicative self-incrementing*", mode it is a positive integer number in the **range from 1 to 100** and represents a **percentage** used to calculate the new size that must be added to the current size every time the buffer needs to grow.
    - ***NOTE_01:*** As expected, in "***f**ixed*" mode, once defined the size, the **buffer size cannot increase**.
- ***mode*** is an operational mode indicator. It can be set to three different integer numbers: **1**, **0**, and **-1**. These values are represented by constants in `buffer.h` (see (***FIXMODE***, ***ADDMODE***, ***MULMODE***). The mode is set when a new buffer is created and cannot be changed later.
    - Constant **0** (***FIXMODE***) indicates that the buffer operates in "***f**ixed-size*" mode.
    - Constant **1** (***ADDMODE***) indicates "***a**dditive self-incrementing*" mode
    - Constant **-1** (***MULMODE***) indicates "***m**ultiplicative self-incrementing*" mode.
- ***addCOffset*** is the distance (measured in chars) from the beginning of the character array (***content***) to the location where the next character is to be added to the existing buffer content.

- ***NOTE_02*****:** ***addCOffset*** must never be larger than ***size***, or else, you are overrunning the buffer in memory and your program may crash at run-time or destroy data.

❖ ***getCOffset*** is the distance (measured in chars) from the beginning of the character array (***content***) to the location of the character which will be returned if the function ***bGetCh()*** is called.

- ***NOTE_03:*** The value ***getCOffset*** must never be larger than ***addCOffset***, or else, you are overrunning the buffer in memory and your program may get wrong data or crash at run-time. If the value of ***getCOffset*** is equal to the value of ***addCOffset***, the buffer has reached the end of its current content.

❖ ***markOffset*** is the distance (measured in chars) from the beginning of the character array (***string***) to the location of a ***mark***.

- ***NOTE_04:*** A *mark* is a location in the buffer that indicates the position of a specific character (for example, the beginning of a word or a phrase). It will be used in the Scanner, but the implementation must be done in the Buffer.

❖ ***flags*** is a field containing different flags and indicators.

Note 2: Remembering Flags

In cases when storage space must be as small as possible, the common approach is to pack several data items into single variable; one common use is a set of single-bit or multiple-bit flags or indicators in applications like compiler buffers, file buffers, and database fields.

*The **flags** are usually manipulated through different **bitwise** operations using a set of "**masks**." Alternative technique is to use bit-fields.*

Using bit-fields allows individual fields to be manipulated in the same way as structure members are manipulated.

*Since almost everything about bit-fields is implementation-dependent, this approach should be avoided if the portability is a concern. In this implementation, you are to use **bitwise** operations and **masks**.*

- Each *flag* or *indicator* uses one or more bits of the ***flags*** field. The flags usually indicate that something happened during a routine operation, such as end of file "eof", end of buffer "eob", integer arithmetic sign overflow, and so on. Multiple-bit indicators can indicate, for example, the mode of the buffer (three different combinations – therefore 2-bits are needed). In this implementation the ***flags*** field has the following structure:

Bit	**MSB 15**	14	13	12	11	10	9	8	7	6	5	4	3	2	1	**0 LSB**
Content	X	X	1	1	1	1	1	1	1	1	1	1	1	1	1	1
Description	eob flag	r flag	Ignored													

- Note that the sequence of 0011…111 (2 bytes) correspond to the value **3FFF** hexadecimal (see `buffer.h`).
- The **MSB 0** of the ***flags*** field is a single-bit *end-of-buffer* (***eob_flag***) flag.
- The ***eob*** bit is by default **0**, and **when set to 1, it indicates that the end of the buffer content has been reached** during the buffer read operation (***bGetCh()*** function). If ***eob_flag*** is set to **1**, the function ***bGetCh()*** should not be called before the ***getCOffset*** is reset by another operation.
- The **MSB 1** of the ***flags*** field is a single-bit reallocation flag (***r_flag***). The ***r_flag*** bit is by default **0**, and **when set to 1, it indicates that the location of the buffer character array in memory has been changed due to memory reallocation**. This could happen when the buffer needs to expand or shrink. The flag can be used to avoid dangling pointers when pointers instead of offsets (positions) are used to access the information in the character buffer.
- ***NOTE_05:*** The rest of the bits are resetting ***eob_flag*** or ***r_flag*** reserved for further use and **must be set by default to 1**. When setting or they must not be changed by the bitwise operation manipulating bit 0 and 1.

❖ In this assignment you need to complete the coding for a "**buffer utility**", which includes the buffer data structure and the associated functions, following strictly the given specifications. Use the data declarations and function prototypes given below.

- ***IMPORTANT NOTE_01:*** **Do not change the names and the data types of the functions and the variables. Any change will be regarded as a serious specification violation.**

1.2. BUFFER HEADER FILE (`buffer.h`)

❖ Check all TODO labels related to activities:

- TODO101: Adjust file header,
- All constant definitions, data type and function declarations (prototypes) must be located in a header file named **`buffer.h`**,
- To constants definition, see from TODO102 to TODO118.
  - **TIP_01**: You need to read the following section (about functional specification) to adjust the values appropriately.
- In the TODO119 include all function declarations that will be used in buffer.

❖ To name a constant you must use ***#define*** preprocessor directive (see `buffer.h`).

❖ ***IMPORTANT NOTE_02:*** The incomplete **`buffer.h`** is posted on Brightspace (BS). All function definitions must be stored in a file named **`buffer.c`**.

1.3. BUFFER FUNCTIONALITIES (`buffer.c`)

You need to implement the following set of buffer utility functions (operations). All function definitions must be stored in a file named **`buffer.c`**. Later, they will be used by all other parts of the compiler when a temporary storage space is needed.

The **first implementation step** in all functions must be the validation (if possible and appropriate) of the function arguments. If an argument value is invalid, the function must return immediately an appropriate failure indicator.

Note 3: Programming Best Practices

In Compilers, you are invited to show your best practices. Some of they are already illustrated here:

- Standard for codification: here, we are using Camel syntax to make code more readable. Especially, when we are not using OO paradigm, the construction of methods and variables should be understood by anyone.

- Boundary conditions: all codes should consider problem (normally during their initialization = parameter checking). See several “Error conditions” shown below.

*- Use **DEFENSIVE PROGRAMMING**: Test not only the initial conditions, but also rules from the specifications carefully.*

- In the **TODO201**, remember to adjust file header.

- ***bPointer bCreate(short size, char increment, char mode)***
  - **TODO202** (function header) and **TODO203** (function definition).
  - This function creates a new buffer in memory (on the program heap), trying to allocate memory for one ***buffer*** structure using ***calloc()***;
  - It tries to allocates memory for one dynamic character buffer (***content***) calling ***malloc()*** with the given initial capacity ***(size)***;
  - This function also copies the given ***size*** value into the Buffer structure ***size*** variable;
  - ***NOTE_06:*** The range of the parameter ***size*** must be between ***0*** and the ***MAXIMUM ALLOWED POSITIVE VALUE*** *– 1* inclusive.
    - The *maximum allowed positive value* is determined by the data type of the parameter ***size***.
  - ***NOTE_07:*** The range of the parameter ***increment*** can vary according to the mode:
    - For ***FIXMODE***, the increment must be always **0**,

        - For ***ADDMODE*** the values acceptable are from the range **1** to **255**,
        - For ***MULMODE***, the values go from **1** to ***MAX_INCREMENT*** (**100**) inclusive.
    - If the ***size*** is **0,** the function tries to create a character buffer with ***size*** adjusted to ***DEFAULT_SIZE*** (**200**) characters and ***increment*** to ***DEFAULT_INCREMENT*** **(15)**.
    - According to the mode parameter (char), the buffer mode is set (use the constant definitions for this): ***FIXMODE*** (**0**)**,** ***ADDMODE*** (**1**)**,** ***MULMODE*** (**-1**)**.**
        - In the ***FIXMODE***, there is no increment applicable.
    - With the correct size, the content of buffer (char*) can be created using ***malloc()***.
    - This function also sets the Buffer structure operational mode indicator ***mode*** and the ***increment***.
    - It also sets the ***flags*** field to its default value which is **3FFF** hexadecimal (**0011.1111.1111.1111** in binary).
    - Finally, on success, the function returns a pointer to the **Buffer** structure. It must return **NULL** pointer on any error which violates the constraints imposed upon the buffer parameters or prevents the creation of a working buffer.
    - ***Error Condition_01:*** If run-time error occurs, the function must return immediately after the error is discovered. Check for all possible errors which can occur at run time. Do not allow **“memory leaks”, “dangling”** pointers, or “**bad**” parameters.

- ***bPointer bAddCh(bPointer const pBuffer, char ch)***
    - **TODO204** (function header) and **TODO205** (function definition).
    - This function is responsible to include a char in the buffer. Because of the limit (given by ***size***), several actions should be done before simply include it in the end of the buffer.
    - Using a bitwise operation, the function resets the ***flags*** field ***r_flag*** bit to 0 (RESET) and tries to add the character ***ch*** to the character array of the given ***buffer*** pointed by ***pBuffer***.
    - Before adding a char, it is required to test the limits: it is not possible to include a char when you are at the limit (***MAXIMUM ALLOWED POSITIVE VALUE*** *– 1*).
    - ***NOTE_07:*** If the buffer is operational and it is **not full**, the symbol can be stored in the character buffer. In this case, the function adds the character to the content of the character buffer, increments ***addCOffset*** by 1 and returns.

- ***NOTE_08:*** If the character buffer is **already full**, the function will try to resize the buffer by increasing the current size to a new one. **How the size is increased depends on the current operational mode of the buffer**.
  - If the operational mode is ***FIXMODE***, the function returns NULL (remember that it is not possible to increase buffer in FIXED mode)
  - If the operational mode is ***ADDMODE***, it tries to increase the current size of the buffer to a *new size* by adding ***increment*** (converted to bytes) to ***size***.
    - If the result from the operation is positive and does not exceed the ***MAXIMUM ALLOWED POSITIVE VALUE –1***, the function proceeds.
    - If the result from the operation is equal to ***MAXIMUM ALLOWED POSITIVE VALUE***, it assigns the ***MAXIMUM ALLOWED POSITIVE VALUE –1*** to the *new size* and proceeds.
      - ***NOTE_09:*** Remember that the ***MAXIMUM ALLOWED POSITIVE VALUE*** is determined by the data type of the variable, which contains the buffer size.
    - If the result from the operation is **negative**, it returns NULL.
  - If the operational mode is ***MULMODE***, it tries to increase the current size of the buffer to a *new size* in the following manner:
    - If the current size can not be incremented anymore because it has already reached the maximum size of the buffer, the function returns NULL.
    - The function tries to increase the current size using the following formulae:

available space = maximum buffer size – current size
*new increment = available space * inc_factor / 100*
new size = current size + new increment

    - **TIP_01**: To use this formula, check eventual casting to guarantee acceptable values. The value can vary according to the percentage of the current size.
    - The *maximum buffer size* is the ***MAXIMUM ALLOWED POSITIVE VALUE –1****.* If the *new size* has been incremented successfully, no further adjustment of the *new size* is required.
    - ***NOTE_10:*** If as a result of the *calculations*, the *current size* cannot be incremented, but it is still smaller than the ***MAXIMUM ALLOWED POSITIVE VALUE –1***, then the new size is set to the value of ***MAXIMUM ALLOWED –1*** and the function proceeds.
- If the operation is successful, the function performs the following operations:

    - The function tries to expand the character buffer calling ***realloc()*** with the *new size*,
    - If the location in memory of the character buffer has been changed by the reallocation, the function sets ***r_flag*** bit to **1** using a bitwise operation,
    - In short, it adds (appends) the character ***ch*** to the buffer content,
    - It also changes the value of ***addCOffset*** by 1, and saves the newly calculated ***size***,
    - In the end, the function returns a pointer to the **buffer** structure.
  - ***Error Condition_02:*** **The function must return NULL on any error**.
    - Some of the possible errors are indicated above but you must check for all possible errors that can occur at run-time,
    - Do not allow "**memory leaks**". Avoid creating "**dangling pointers**" and using "**bad**" parameters,
    - The function ***must not destroy*** the buffer or the contents of the buffer even when an error occurs – it must simply return NULL leaving the existing buffer content intact.

- ***int bClean(bPointer const pBuffer)***
  - **TODO206** (function header) and **TODO207** (function definition).
  - The function retains the memory space currently allocated to the buffer, but re-initializes all appropriate data members of the given ***buffer*** structure so that the buffer will appear as just created to the client functions (for example, next call to ***bAddCh()*** will put the character at the beginning of the character buffer).
  - The function does not need to clear the existing contents of the character buffer.
  - ***Error Condition_03:*** The function returns **RT_FAIL_1** (**-1**) on failure.

- ***int bFree(bPointer const pBuffer)***
  - **TODO208** (function header) and **TODO209** (function definition).
  - The function de-allocates (frees) the memory occupied by the character buffer and the ***buffer*** structure.
  - ***Error Condition_04:*** The function should not cause abnormal behavior (crash).

- ***int bIsFull(bPointer const pBuffer)***

- TODO210 (function header) and TODO211 (function definition).
- The function returns ***1*** if the character buffer is full; it returns ***0*** otherwise.
- ***Error Condition_05:*** The function returns **RT_FAIL_1** on failure.

❖ ***short bGetAddChOffset(bPointer const pBuffer)***

- TODO212 (function header) and TODO213 (function definition).
- The function returns the current ***addCOffset***.
- ***Error Condition_06:*** The function returns **RT_FAIL_1** on failure.

❖ ***short bGetSize(bPointer const pBuffer)***

- TODO214 (function header) and TODO215 (function definition).
- The function returns the current size of the character buffer.
- ***Error Condition_07:*** The function returns **RT_FAIL_1** on failure.

❖ ***int bGetMode(bPointer const pBuffer)***

- TODO216 (function header) and TODO217 (function definition).
- The function returns the value of ***mode*** to the calling function.
- ***Error Condition_08:*** If a run-time error is possible, the function should notify the calling function about the failure.

❖ ***int bGetMarkOffset (bPointer const pBuffer)***

- TODO218 (function header) and TODO219 (function definition).
- The function returns the value of ***markOffset*** to the calling function.
- ***Error Condition_09:*** The function returns **RT_FAIL_1** on failure.

❖ ***short bSetMarkOffset(bPointer const pBuffer, short mark)***

- TODO220 (function header) and TODO221 (function definition).
- The function sets ***mark*** to ***markOffset***.
- The parameter ***mark*** must be within the current limit of the buffer (0 to ***addCOffset*** inclusive).
- ***Error Condition_10:*** The function returns **RT_FAIL_1** on failure.

- ***bPointer bFinish(bPointer const pBuffer, char ch)***
  - **TODO222** (function header) and **TODO223** (function definition).
  - For all operational modes of the buffer, the function shrinks (or in some cases may expand) the buffer to a *new size*.
  - The *new size* is the current limit plus a space for one more character. In other words, the *new size* is ***addCOffset* + *1*** converted to bytes.
  - The function uses ***realloc()*** to adjust the *new size*, and then updates all the necessary members of the buffer descriptor structure.
  - Before returning a pointer to ***Buffer***, the function adds the ***ch*** to the end of the character buffer and increments ***addCOffset.***
  - ***NOTE_11:*** It must set the ***r_flag*** bit appropriately***.***
  - **TIP_02**: This function has several similarities with the logic performed in the ***bAddCh***() function.
  - ***Error Condition_11:*** The function must return NULL if for some reason it cannot to perform the required operation.

- ***int bDisplay(bPointer const pBuffer, char nl)***
  - **TODO224** (function header) and **TODO225** (function definition).
  - This function is intended to print the content of buffer (in ***content*** field).
  - Using the ***printf()*** library function, the function prints character by character the contents of the character buffer to the standard output (***stdout***).
  - In a loop, the function prints the content of the buffer calling ***bGetCh()*** and **checking the flags** in order to detect the **end of the buffer** content (using other means to detect the end of buffer content will be considered a significant specification violation).
  - After the loop ends, it checks the ***nl*** and if it is not **0,** it prints a new line character.
  - Finally, it returns **the number of characters printed**.
  - ***Error Condition_12:*** The function returns **RT_FAIL_1** on failure.

- ***int bLoad(bPointer const pBuffer, FILE* const fi)***
  - **TODO226** (function header) and **TODO227** (function definition).

- The function loads (reads) an open input file specified by ***fi*** into a buffer specified by ***pBuffer***. The file is supposed to be a plain file (for instance, a source code).
- **TIP_03**: The function must use the standard function ***fgetc(fi)*** to read one character at a time and the function ***bAddCh()*** to add the character to the buffer.
- ***NOTE_12:*** If the current character cannot be added to the buffer (***bAddCh()*** returns NULL), the function returns the character to the file stream (file buffer) using ***ungetc()*** library function and then returns **-2** (use the defined ***LOAD_FAIL*** constant). The operation is repeated until the standard macro ***feof(fi)*** detects end-of-file on the input file. The end-of-file character must not be added to the content of the buffer.
- Only the standard macro ***feof(fi)*** must be used to detect end-of-file on the input file.
- Using other means to detect end-of-file on the input file will be considered a significant specification violation.
- ***Error Condition_13:*** If some other run-time errors are possible, the function should return **–1**. If the loading operation is successful, the function must return the number of characters added to the buffer.

❖ ***int bIsEmpty(bPointer const pBuffer)***

- **TODO228** (function header) and **TODO229** (function definition).
- If the ***addCOffset*** is 0, the function returns 1. Otherwise it returns 0.
- ***Error Condition_14:*** The function returns **RT_FAIL_1** on failure.

❖ ***char bGetCh(bPointer const pBuffer)***

- **TODO230** (function header) and **TODO231** (function definition).
- This function is used to read the buffer. The function performs the following steps:
  - In the end, it increments ***getCOffset*** by **1** and returns the character located at ***getCOffset***.
  - If ***getCOffset*** and ***addCOffset*** are equal, it sets the ***flags*** field ***eob_flag*** bit to **1** using a bitwise operation and returns number **0**; otherwise, it sets ***eob_flag*** to **0** using a bitwise operation,
- ***Error Condition_15:*** The function returns **RT_FAIL_1** on failure.

- ***int bRewind(bPointer const pBuffer)***
    - **TODO232** (function header) and **TODO233** (function definition).
    - The function set the ***getCOffset*** and ***markOffset*** to 0, so that the buffer can be re-read again.
    - ***Error Condition_16:*** The function returns **RT_FAIL_1** on failure.

- ***bPointer bRetract(bPointer const pBuffer)***
    - **TODO234** (function header) and **TODO235** (function definition).
    - The function decrements ***getCOffset*** by **1**.
    - Once decremented, the buffer is returned.
    - ***Error Condition_17:*** The function returns ***NULL*** on failure.

- ***short bRestore(bPointer const pBuffer)***
    - **TODO236** (function header) and **TODO237** (function definition).
    - The function sets ***getCOffset*** to the value of the current ***markOffset***.
    - ***Error Condition_18:*** If a run-time error is possible, it should return **-1**; otherwise, it returns ***getCOffset***.

- ***short bGetChOffset(bPointer const pBuffer)***
    - **TODO238** (function header) and **TODO239** (function definition).
    - The function returns ***getCOffset*** to the calling function.
    - ***Error Condition_19:*** The function returns **RT_FAIL_1** on failure.

- ***size_t bGetIncrement(bPointer const pBuffer)***
    - **TODO240** (function header) and **TODO241** (function definition).
    - The function returns the non-negative value of ***increment*** to the calling function. The return datatype (***size_t***) must be appropriately checked according to datatype you are getting.
    - ***Error Condition_20:*** If a run-time error is possible, the function should return a special value: ***RT_INC_FAIL*** (**0x100**).

- ***char* bGetContent(bPointer const pBuffer, short pos)***

?
short chPosition
NOT

- **TODO242** (function header) and **TODO243** (function definition).
- The function returns a pointer to the location of the character buffer indicated by ***pos*** that is the distance (measured in chars) from the beginning of the character array (***content***).
- ***Error Condition_21:*** If a run-time error is possible, it should return **NULL**.

❖ ***short bufferAddCPosition (bPointer const pBuffer)*** A (bStructure* . . . . .)

- **TODO244** (function header) and **TODO245** (function definition).
- The function returns ***getCOffset*** to the calling function.
- ***Error Condition_19:*** The function returns **RT_FAIL_1** on failure.

❖ ***unsigned short bGetFlags(bPointer const pBuffer)***

- **TODO246** (function header) and **TODO247** (function definition),
- The function returns the flag field from buffer,
- ***Error Condition_22:*** The function returns **RT_FAIL_1** on failure,
- ***NOTE_13:*** It is suggested (bonus) the development of the macro function to define the bGetFlags (**TODO248**).

Task 2: Testing the Buffer

2.1. GENERAL VIEW

To test your program, you are to use the test harness program ***evalBuffer.c*** **(avoid change it)** and test it with input files.

❖ For standard tests: The input files **a1e.pls** (an empty file), and **a1r.pls** (a correct PLATYPUS file) will be provided.

❖ For additional tests, the input file (**a1bigf.pls**) is also provided.

❖ The corresponding output files are:

- When processing a1e.pls:
  - ***a1e.out***: For empty file (using any mode, for instance, ***FIXMODE***);
- When processing a1r.pls:

    - ***a1fi.out*** (for ***FIXMODE***). ***a1ai.out*** (for ***ADDMODE***), ***a1mi.out*** (for ***MULMODE***).

- Those files are generated by my test program and contain plain ASCII text.
- They are available as part of the assignment postings on Brightspace (BS).
- ***IMPORTANT NOTE_03:*** **You must create a standard console project named *buffer.exe* with an executable target *buffer* (see *Creating_C_Project* document in Lab0).**

- In order to test the execution (and checking the requirements mentioned in the A1_Marking_Sheet), it is necessary to generate a binary that can accept the following parameters: **input file**, **mode**, **size** and **increment**.

The **standard tests** are composed by the following parameters in the execution:

- `buffer.exe a1e.pls f 0 0`
- `buffer.exe a1r.pls f 0 0`
- `buffer.exe a1r.pls a 0 0`
- `buffer.exe a1r.pls m 0 0`

The **additional tests** are composed by the following parameters in the execution:

```
• buffer.exe a1bigf.pls a  200   15
• buffer.exe a1bigf.pls a  200   128
• buffer.exe a1bigf.pls a  32766 2
• buffer.exe a1bigf.pls f  200   0
• buffer.exe a1bigf.pls f  200   128
• buffer.exe a1bigf.pls m  200   15
• buffer.exe a1bigf.pls m  32757 9
• buffer.exe a1bigf.pls m  32756 15
• buffer.exe a1bigf.pls a  200   0
• buffer.exe a1bigf.pls m  1     100
• buffer.exe a1bigf.pls a  0     1
• buffer.exe a1bigf.pls f  32767 0
• buffer.exe a1bigf.pls m  200   101
• buffer.exe a1bigf.pls f  0     1
• buffer.exe a1bigf.pls f  -1    0
```

TIP1

These executions must generate files in the output. For instance:

```
buffer.exe a1e.pls f 0 0 > a11e.seva
```

This is especially important because the final comparison is done by files (not only by visual checking), for instance, comparing the following outputs:

```
fc /b a11e.sout a11e.seva
```

So, it is required to be sure that files are being generated ok and the outputs can be checked using binary comparison.

TIP2:

There is a very useful program you should download and install on your computer: **Total Commander**. This is very useful file and program manager (Window Explorer is supposed to be such a program). It is also a very powerful FTP client and has built-in zip/unzip utilities. And it could be very helpful when you need to compare.

2.2. ABOUT EXECUTION

Here is a brief description of the program that is provided for you on Brightspace (BS).

- It simulates "normal" operating conditions for your buffer utility.
- The program (***evalBuffer.c***) main function takes two parameters from the command line:
  - An input file name and a character (**f** – ***FIXMODE***, **a** – ***ADDMODE***, or **m** – ***MULMODE***) specifying the buffer operational mode. It opens up a file with the specified name (for example, ***a1r.pls***), creates a buffer (calling ***bCreate()***), and loads it with data from the file using the ***bLoad()*** function.
  - Then the program prints several buffer info in the ***displayBuffer***() function, including the current size (***bGetSize()***), the current size (***bGetAddOffset()***), the current operational mode (***bGetMode()***), the increment factor (***bGetIncrement()***), the first symbol (using ***bGetContent()***) and the flags (***bGetFlags()***). After this, the program rewinds (***bRewind()***) in order to print the contents of the buffer (calling ***bDisplay()***).
  - Finally, it sets the end of the buffer (***bFinish()***), printing again and freeing it before terminate (***bFree()***).
- ***NOTE:*** Your program must not overflow any buffers in any operational mode, no matter how long the input file is.
- **TIP**: **The provided main program will not test all your functions.** You are strongly encouraged to test all your buffer functions with your own test files and modified main function.

Task 3: Bonus Task

You have a chance to get bonus by **implementing a Preprocessor Macro Definition and Expansion (1%)**

- **TASK:** Implement ***bGetFlags()*** both as a function and a macro.
- Using conditional processing, you must allow the user to choose between using the macro or the function in the compiled code.
    - If **FLAGS** name is defined, the macro should be used in the compiled code.
    - If the **FLAGS** name is not defined or undefined, the function should be used in the compiled code.
- ***IMPORTANT NOTE:*** To receive credit for the bonus task your code must be well documented, tested, and working.

Submission Details

- **Digital Submission: Compress** into a **zip** file the following files*:* `buffer.h`, `buffer.c` and additional files related to submission - your additional input/output test files if you have any. Also include a Cover page and a Test Plan. Check the A1 Marking Sheet for it, as well as Submission Standard document.

The submission must follow the course submission standards. You will find the *Assignment Submission Standard* as well as the *Assignment Marking Guide* (

- **Submission Standard and Marking Guide.pdf)** for the Compilers course on the Brightspace.
- Upload the zip file on Brightspace. The file must be submitted prior or on the due date as indicated in the assignment.
- ***IMPORTANT NOTE:*** The name of the file must be **Your Last Name** followed by the last three digits of your student number followed by your lab **section number**. For example: **Sousa123_s10.zip**.
    - If you are working in teams, please, include also your partner. For instance, something like: **Sousa123_Melo456_s10.zip**.
    - **Remember:** Only students from the **same section** can constitute a specific team.
- ***IMPORTANT NOTE:*** **Assignments will not be marked if there are not source files in the digital submission.** Assignments could be late, but the lateness will affect negatively your mark: see the Course Outline and the Marking Guide. All assignments must be successfully completed to receive credit for the course, even if the assignments are late.
- **Evaluation Note:** Make your functions as efficient as possible.
    - These functions are called many times during the compilation process.

- The functions will be graded with respect to design, documentation, error checking, robustness, and efficiency. When evaluating and marking your assignment, I will use the standard project and **`evalBuffer.c`** and the test files posted on Brightspace.
- If your program compiles, runs, and produces correct output files, it will be considered a ***working program***.
- Additionally, I will try my best to "crash" your functions using a modified main program, which will test all your functions including calling them with "invalid" parameters.
- I will use also some additional test files (for example, a large file). So, test your code as much as you can!
- This can lead to fairly big reduction of your assignment mark (see **Submission Standard and Marking Guide** and **MarkingSheetA1** documents).

"It is part of the nature of humans to begin with romance (buffer) and build to reality (compiler)" (Ray Bradbury)

File update: Jan 14th, 2021.

Check the Assignment 1 before going to next Assignments!